REPAST STATECHARTS GUIDE

JONATHAN OZIK, NICK COLLIER - REPAST DEVELOPMENT TEAM

0. Before we Get Started

Before we can do anything with Repast Simphony, we need to make sure that we have a proper installation of Repast Simphony 2.1. Instructions on downloading and installing Repast Simphony on various platforms can be found on the Repast website.

1. Getting Started with Statecharts

Agent states and transitions between states are an important abstraction in agent-based modeling. While it is possible for Repast Simphony users to create their own implementation of state-based agent behaviors (e.g., by adapting the State pattern in Gamma et al. 1994[1]) and even agent state visualizations, the effort involved in doing so is usually prohibitive. By integrating an agent statecharts framework into Repast Simphony, we made it easy for users of all levels to take advantage of this important modeling paradigm. Statecharts are visual representations of states and the transitions between those states[2]. Statecharts can be very effective in visually capturing the logic within agents and quickly conveying the underlying dynamics of complex models.

Figure 1 shows a simple example of a statechart created with the Repast Simphony statecharts framework. The logic embedded in the diagram is mapped directly to the execution logic of an agent-based model. The benefits of the Repast Simphony statecharts framework include: improved clarity of a model's logic for model design, improved turnaround times for developing complex state based agent models, and the ability to convey in a compelling manner the internal state of agents as a simulation evolves to both experienced agent-modelers and to non-modelers alike. In the rest of this guide we will present the Repast Simphony statecharts framework.

1.1. **Adding Statecharts.**

A statechart can be added to any Java, Groovy or ReLogo class. Right-clicking the class of interest and selecting New → Statechart Diagram (Figure 2) will bring up the new statechart wizard (Figure 3). The editable new statechart wizard elements are:

Date: June 29, 2013.

[1]Gamma, E., R. Helm, R. Johnson, and J. M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. illustrated ed. Addison-Wesley Professional, 1994.

[2]Statecharts were first proposed by Harel in Harel, D., 1987. Statecharts: A visual formalism for complex systems. Sci. Comput. Program., 8(3), pp.231274.

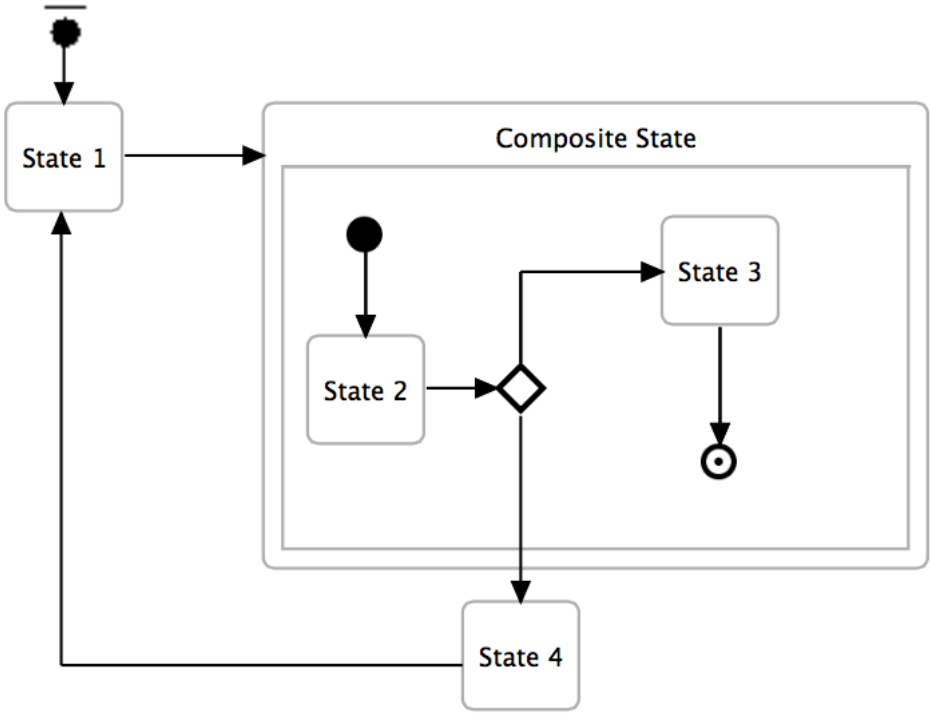

FIGURE 1. An example statechart created with the Repast Simphony visual statecharts editor.

File Name: This is the .rsc statechart file name that will be edited with the visual statecharts editor.
Name: The display name of the statechart.
Class Name: The name of the statechart class which will be generated.
Package: The package name within the `src-gen` source folder where the statechart class source code will be generated.
Agent Class: This is the agent class that will be associated with the statechart.

After accepting or modifying the defaults, clicking the *Finish* button will bring up the newly created blank statechart editor.

FIGURE 2. Creating a new statechart.

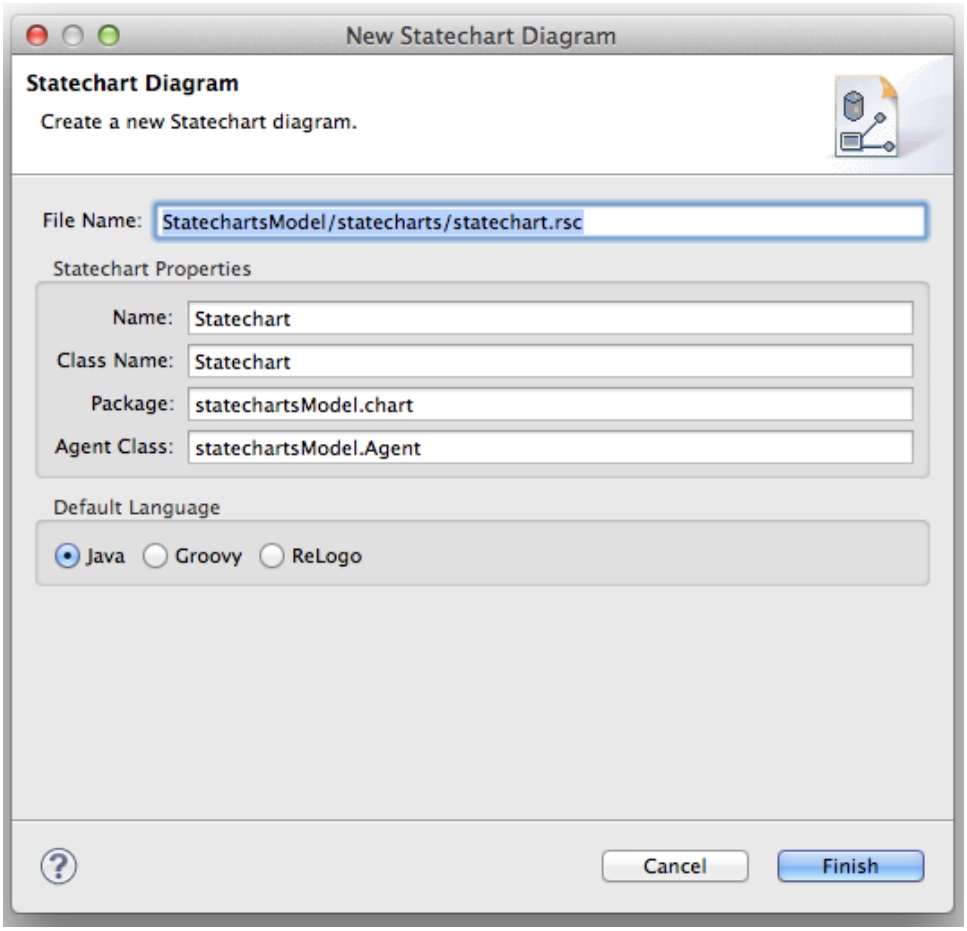

FIGURE 3. New statechart wizard.

1.2. **Statecharts Editor.** The statecharts editor is divided into three main panels (Figure 4). The first (Figure 4a) is the statecharts workspace area. This is where the visual elements of a statechart are created and arranged. The palette panel (Figure 4b) shows the available statechart elements that can be used in the statecharts workspace[3]. Clicking on an element in the palette panel and then clicking on the statecharts workspace will create an instance of that element in the workspace. The properties panel (Figure 4c) shows the properties of the element selected in the statecharts workspace. If, as in Figure 4, no element is selected, the properties of the statechart itself are shown. In addition to displaying element properties, the panel is also where the properties of elements can be edited. A statechart has a priority which indicates the order in which it will be resolved with respect to other statecharts (see red box in Figure 5). So, for example, if an agent has two statecharts (A and B) and it is always desired that statechart A should be resolved before statechart B, giving statechart A a higher priority will ensure this.

There is a contextual menu approach for adding elements to the workspace as well. Simply hovering over an area in the workspace will reveal a contextual menu of the available elements appropriate for the region pointed to. If the mouse pointer is on an empty space in the workspace background, you will see the contextual menu in Figure 6. If the pointer is on a state, you will see transitions shortcuts like in Figure 7, the left symbol indicating a connection from this state and the right symbol a connection to this state. Finally, if the pointer is inside a composite state, you will see the contextual menu in Figure 8.

[3]The detailed explanations for these elements will be covered in Sections 2 and 3.

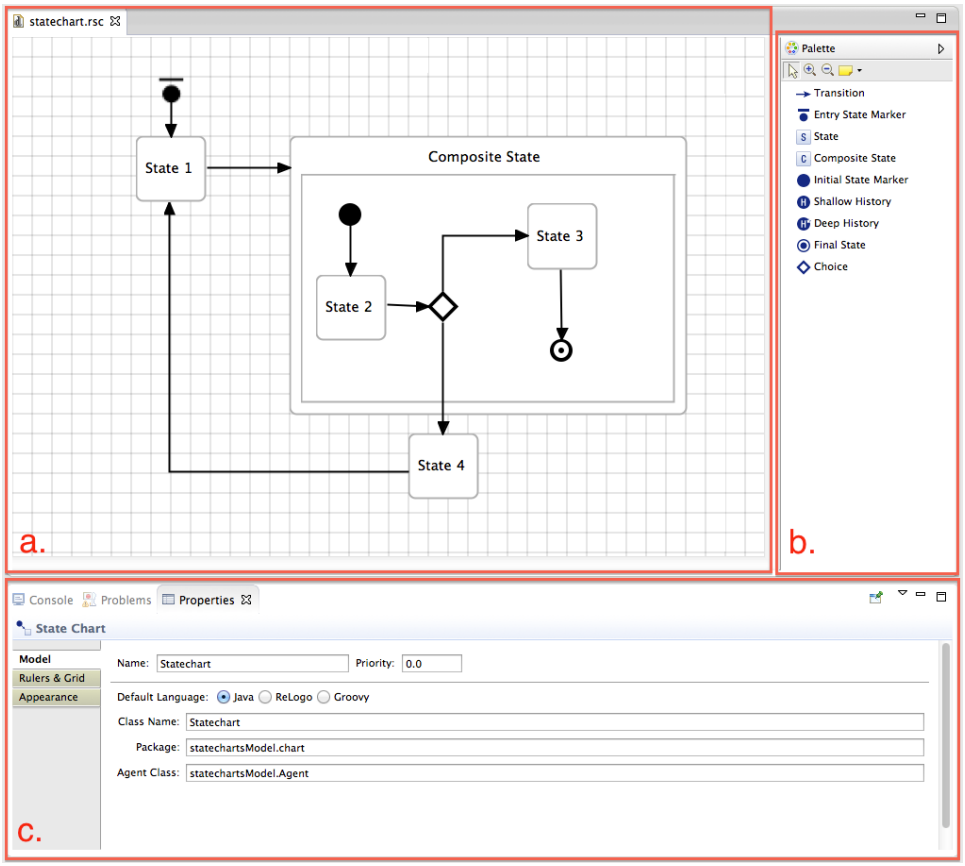

FIGURE 4. The components of the statecharts visual editor. a) The workspace area where the statechart elements are created and arranged. b) The palette panel of available elements, including selection, zoom and note tools. c) The properties panel of the selected statechart element in the workspace (a). If no element is selected, the properties of the statechart itself are displayed.

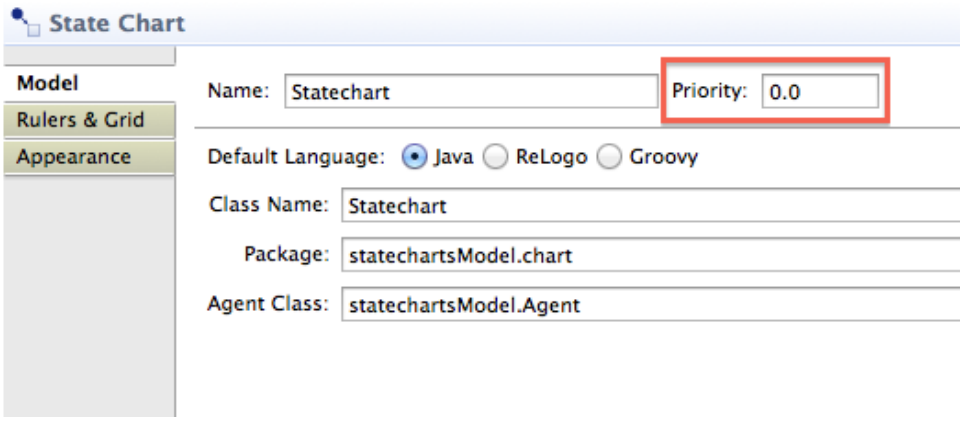

FIGURE 5. The statechart properties panel with the priority element indicated by a red box.

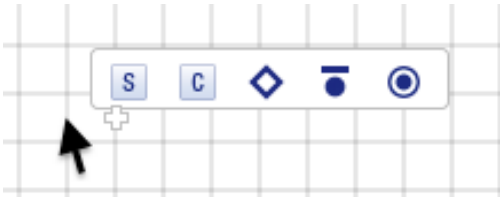

FIGURE 6. The default contextual menu when the pointer is on a blank area of the workspace.

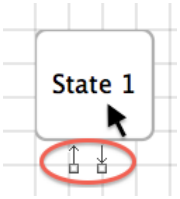

FIGURE 7. The transitions shortcuts, circled in red, for making connections from (left) and to (right) the state.

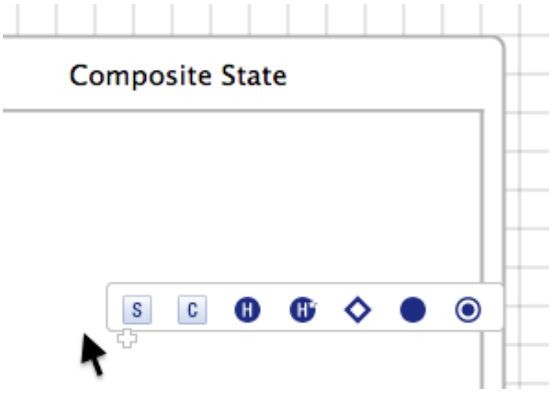

FIGURE 8. The contextual menu when the pointer is inside a composite state.

FIGURE 9. Simple state.

2. States

One of the fundamental building blocks of statecharts are states. Here we introduce the different types of states that exist within the Repast Simphony statecharts framework.

2.1. Entry State Marker.

Every statechart must have an entry state marker. This defines the path through which the statechart begins when it is activated.

2.2. Simple State.

A simple state looks like Figure 9. At any one point in time within an active statechart, one and only one of the simple states will be active. In addition to their *ID*, simple states can have *On Enter* and *On Exit* actions defined, as seen in the simple state properties panel in Figure 10. These actions are triggered when entering or exiting the simple state, respectively. The keywords available within the two action blocks are:

agent: This is the agent that contains the statechart. Any method (e.g., `customMethod`) defined on the agent can be invoked through this reference (e.g., `agent.customMethod()`).

state: This is the state itself. For example, the state's *ID* can be accessed via `state.getId()`.

params: This is the model's `Parameters` object. As an example, a double valued parameter `dParam` can be retrieved with: `params.getDouble("dParam")`[4].

As is the case with all types of action blocks, their logic can be specified using Java, Groovy or ReLogo. Specifically, any Java, Groovy or ReLogo code can be used to express the behavior that should be executed upon entry to or exit from the state[5].

[4]See the source or JavaDoc for `repast.simphony.parameter.Parameters` for all of the available methods.

[5]When using the ReLogo option, the `agent` parameter is implicit so writing `customMethod()` is equivalent to `agent.customMethod()`.

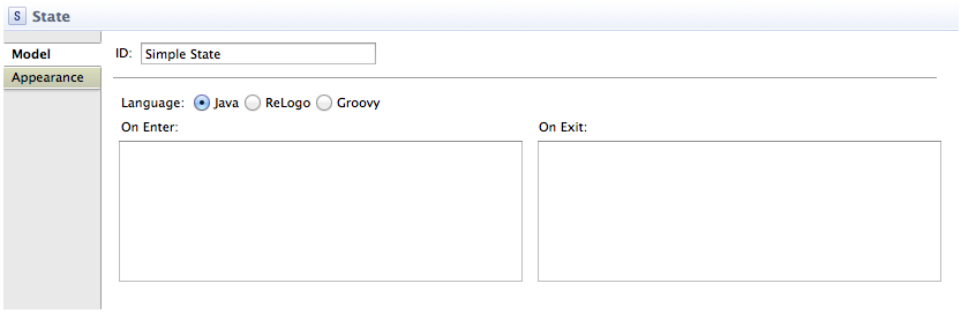

FIGURE 10. Simple state properties.

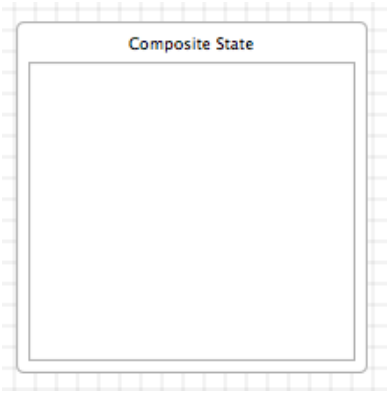

FIGURE 11. Composite state.

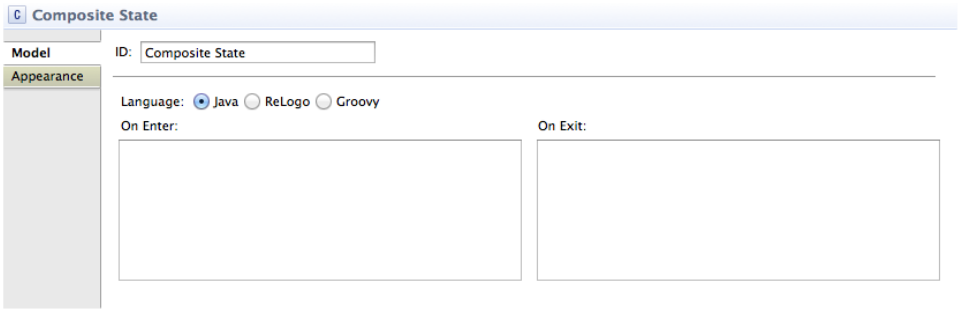

FIGURE 12. Composite state properties.

2.3. **Composite State.** C

Composite states are used to nest elements within a statechart. Figure 11 shows an empty composite state. Composite states can include the following elements:

- Simple state (Section 2.2)
- Composite state (Section 2.3)
- Initial state marker (Section 2.4)
- History state (Section 2.5)
- Final state (Section 2.6)
- Branching state (Section 2.7)

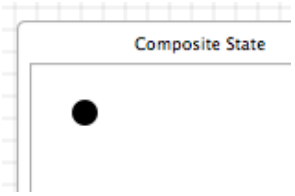

FIGURE 13. Initial state marker (within a composite state).

2.4. **Initial State Marker.**

2.5. **History State.**

2.6. **Final State.**

2.7. **Branching State.**

3. TRANSITIONS

Transitions between states make up the other type of fundamental building block of statecharts. In this section we introduce the different types of transitions that can be used within the Repast Simphony statecharts framework.

There are two overall types of transitions, *regular transitions* (Figure 15) which connect different states and *self transitions* (Figure 16) which are internal to a state[6]. There are a number of different transition trigger types, demonstrated in the transition properties panel in Figure 14 (these will be discussed below in further detail).

For any transition an *On Transition* action can be defined. This action will be executed whenever the transition is traversed. The keywords available within the *On Transition* action block are:

agent: This is the agent that contains the statechart.
transition: This is the transition itself. For example, the transition's source state can be accessed via: `transition.getSource()`[7].
params: This is the model's `Parameters` object.

For almost all types of transitions[8] a *Guard* condition can be defined. A *Guard* condition is an additional boolean condition that has to be satisfied for a transition that is valid to be actually considered as a candidate for traversal. This condition is specified by a block

[6]One also has the ability to define a *regular transition* that begins and ends at the same state. Unlike the *self transition* case, each time the *regular transition* is taken, the state will be exited and subsequently re-entered.

[7]See the source or JavaDoc for `repast.simphony.statecharts.Transition` for all of the available methods.

[8]All transitions except default transitions out of branching states.

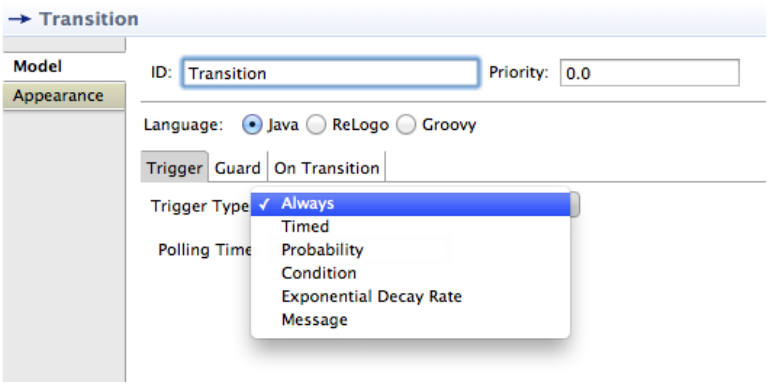

FIGURE 14. The properties panel showing the different types of transitions that are available.

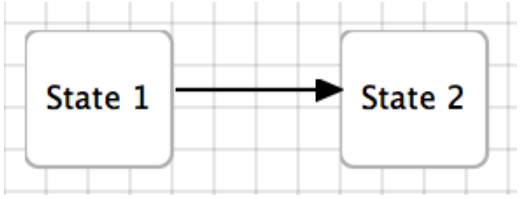

FIGURE 15. Regular transition between states 1 and 2.

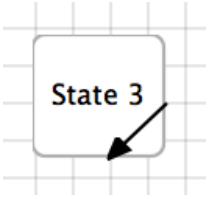

FIGURE 16. Self transition internal to State 3.

of code that returns a boolean and the keywords available in a *Guard* condition are the same as those in an *On Transition* action block.

When there is more than one valid transition ties are broken using the priority of the transition. If the priorities of valid transitions are equal then one of the transitions will be chosen with a uniform random probability. The priority of a transition can be specified in the transition's properties panel.

Regular transitions can be divided into zero time transitions and non-zero time transitions[9]. For zero time transitions when a new state is entered, if there is a valid zero time transition out of it, that transition is followed immediately. Always (Section 3.1), Condition (Section 3.4) and Message (Section 3.6) transition triggers are zero-time transitions.

Every transition has a *polling time* associated with it. This indicates the frequency (in simulation *tick* units) with which the transition is considered for validity.

Next we present the different transition trigger types in more detail.

[9]All *self transitions* are non-zero time transitions.

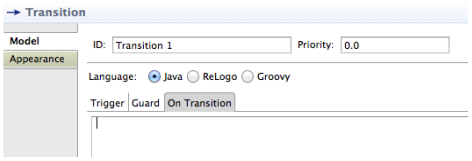

FIGURE 17. Properties panel for a transition showing the *On Transition* action block.

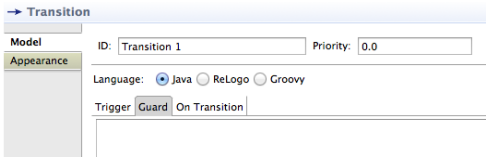

FIGURE 18. Properties panel for a transition showing the *Guard* condition block.

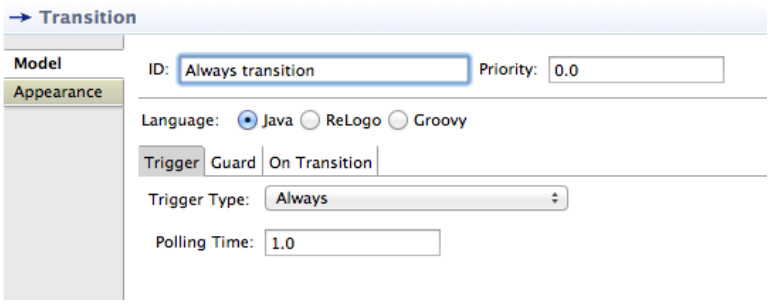

FIGURE 19. The properties panel for an *always* transition.

3.1. **Always Trigger.** *Always* triggers are always valid. The transition can, however, be made invalid by defining a *Guard* condition. Because this trigger type results in zero time transitions, it is important to make sure that there are no *always* trigger transitions contributing to zero time loops in any statechart you create, since this has the potential to create never-ending loops. The properties panel for an *always* trigger transition is in Figure 19. One use for *always* trigger transitions is as self transitions to execute some action at a set polling time.

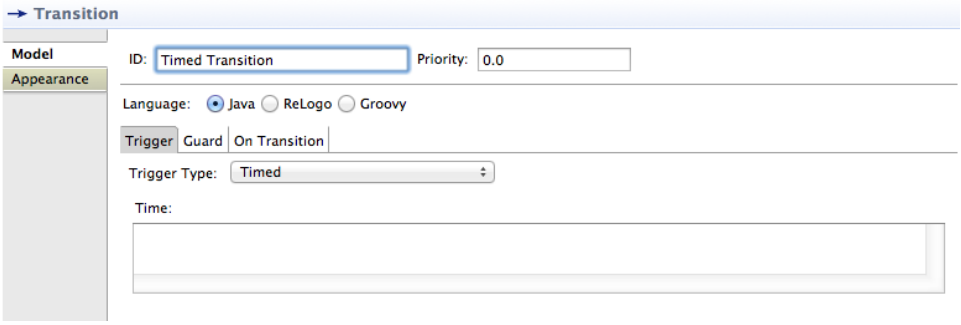

FIGURE 20. The properties panel for a *timed* transition.

3.2. **Timed Trigger.** *Timed* triggers become valid after some time, measured in simulation *ticks*. The properties panel for a *timed* trigger transition is shown in Figure 20. The *Time* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., `2` or `agent.getDelay()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). If at the time a *timed* trigger is valid a *Guard* condition keeps the transition from being valid, the transition does not get reinitialized and will simply remain invalid.

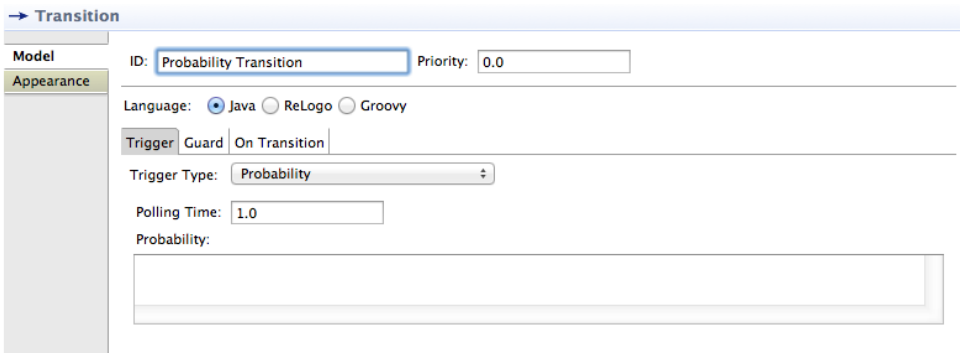

FIGURE 21. The properties panel for a *probability* transition.

3.3. **Probability Trigger.** *Probability* triggers are evaluated as valid with a specified probability. The properties panel for a *probability* trigger transition is shown in Figure 21. The *Probability* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., `2` or `agent.getProbability()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated each time the transition is polled for validity.

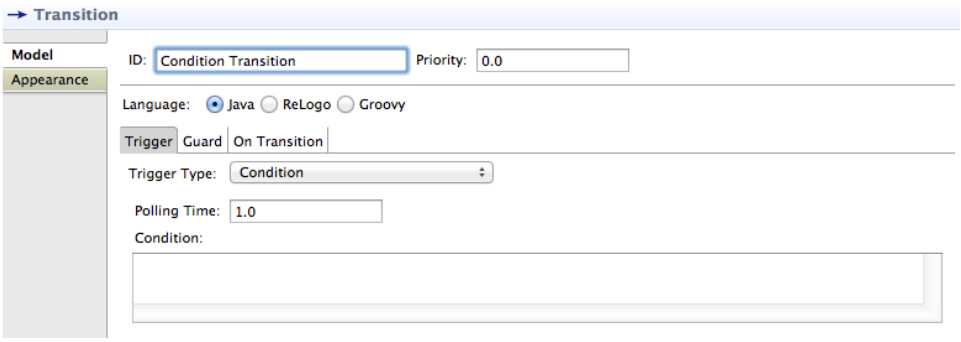

FIGURE 22. The properties panel for a *condition* transition.

3.4. **Condition Trigger.** *Condition* triggers are evaluated as valid based on a specified condition. The properties panel for a *condition* trigger transition is shown in Figure 22. The *Condition* element in the properties panel accepts general Java, Groovy or ReLogo code returning a boolean value, including simple boolean entries (e.g., `true` or `agent.getCondition()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated each time the transition is polled for validity.

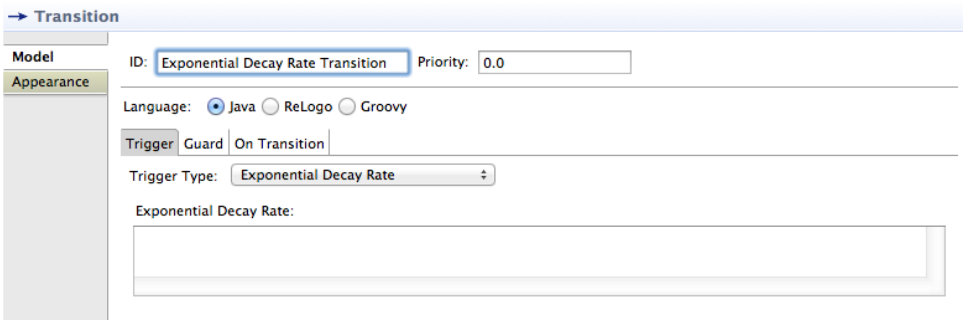

FIGURE 23. The properties panel for a *exponential decay rate* transition.

3.5. **Exponential Decay Rate Trigger.** *Exponential decay rate* triggers become valid after a random time following the exponential distribution. The properties panel for a *exponential decay rate* trigger transition is shown in Figure 23. The *Exponential Decay Rate* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., `2` or `agent.getDecayRate()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated when the transition is initialized (i.e., when a state is entered that has a possible *exponential decay rate* transition leading out of it). The code block supplies the λ parameter to the exponential distribution specified by the probability density function:

$$f(t) = \lambda e^{-\lambda t} \tag{1}$$

The expected value of an exponentially distributed random variable with parameter λ is $1/\lambda$ so given, for example, a λ of 2, the expected value for the time it would take for an *exponential decay rate* transition to trigger would be 0.5 in units of *ticks*.

3.6. **Message Transition.**

3.6.1. *When Message Meets Condition.*

3.6.2. *When Message Equals.*

3.6.3. *When Message is of Class.*

3.6.4. *Always.*